Teach Yourself CORBA in 14 days

Jeremy L. Rosenberger

SAMS PUBLISHING

201 West 103rd Street
Indianapolis, Indiana 46290

To Camilla: For your limitless patience and support while I turned your entire life upside down.
To my parents: Without you, it is improbable that I would ever have had the opportunity to write this book.

FIRST EDITION

International Standard Book Number: 0-672-31208-5

Library of Congress Catalog Card Number: 97-68736

01 00 99 4 3 2

Interpretation of the printing code: The rightmost double-digit number is the year of the book's printing; the rightmost single-digit, the number of the book's printing. For example, a printing code of 98-1 shows that the first printing of the book occurred in 1998.

Composed in AGaramond and MCPdigital by Macmillan Computer Publishing

Printed in the United States of America

Trademarks

President Richard K. Swadley

Publisher Joseph B. Wikert

Managing Editor Jodi Jensen

Indexing Manager Johnna L. VanHoose

Director of Software and User Services Cheryl Willoughby

Brand Director Alan Bower

Acquisitions Editor

Steve Straiger

Development Editor
Tony Amico

Production Editor
Susan Ross Moore

Copy Editor
Kate Talbot

Indexer
Bruce Clingaman

Technical Reviewer
Andrew Watson

Editorial Coordinators
Mandie Rowell
Katie Wise

Team Coordinator
Carol Ackerman

Editorial Assistants
Carol Ackerman
Andi Richter
Rhonda Tinch-Mize
Karen Williams

Cover Designer
Karen Ruggles

Cover Illustration
Eric Lindley

Book Designer
Gary Adair

Copy Writer
David Reichwein

Production Team Supervisor
Andrew Stone

Production Team
Chris Livengood
Shawn Ring

CORBA 1.0

Following the OMG's formation in 1989, CORBA 1.0 was introduced and adopted in December 1990. It was followed in early 1991 by CORBA 1.1, which defined the Interface Definition Language (IDL) as well as the API for applications to communicate with an Object Request Broker (ORB). (These are concepts that you'll explore in much greater detail on Day 2.) A 1.2 revision appeared shortly before CORBA 2.0, which with its added features quickly eclipsed the 1.x revisions. The CORBA 1.x versions made an important first step toward object interoperability, allowing objects on different machines, on different architectures, and written in different languages to communicate with each other.

CORBA 2.0 and IIOP.

CORBA 1.x was an important first step in providing distributed object interoperability, but it wasn't a complete specification. Although it provided standards for IDL and for accessing an ORB through an application, its chief limitation was that it did not specify a standard protocol through which ORBs could communicate with each other. As a result, a CORBA ORB from one vendor could not communicate with an ORB from another vendor, a restriction that severely limited interoperability among distributed objects.

Enter CORBA 2.0. Adopted in December 1994, CORBA 2.0's primary accomplishment was to define a standard protocol by which ORBs from various CORBA vendors could communicate. This protocol, known as the Internet Inter-ORB Protocol (IIOP, pronounced "eye-op"), is required to be implemented by all vendors who want to call their products CORBA 2.0 compliant. Essentially, IIOP ensures true interoperability among products from numerous vendors, thus enabling CORBA applications to be more vendor-independent. IIOP, being the *Internet* Inter-ORB Protocol, applies only to networks based on TCP/IP, which includes the Internet and most intranets.

The CORBA standard continues to evolve beyond 2.0; in September 1997, the 2.1 version became available, followed shortly by 2.2; 2.3 is expected in early 1998. (The OMG certainly is keeping itself busy!) These revisions introduce evolutionary (not revolutionary) advancements in the CORBA architecture.

CORBA Architecture Overview

Finally, having learned the history and reasons for the existence of CORBA, you're ready to examine the CORBA architecture. You'll cover the architecture in greater detail on Day 2, but Day 1 provides you with a very general overview—an executive summary, if you will—of what composes the CORBA architecture.

First of all, CORBA is an object-oriented architecture. CORBA objects exhibit many features and traits of other object-oriented systems, including interface inheritance and polymorphism. What makes CORBA even more interesting is that it provides this capability even

when used with nonobject-oriented languages such as C and COBOL, although CORBA maps particularly well to object-oriented languages like C++ and Java.

New Term: *Interface inheritance* is a concept that should be familiar to Objective C and Java developers. In the contrasting *implementation inheritance,* an implementation unit (usually a class) can be derived from another. By comparison, interface inheritance allows an interface to be derived from another. Even though interfaces can be related through inheritance, the implementations for those interfaces need not be.

The Object Request Broker (ORB)

Fundamental to the Common Object Request Broker Architecture is the Object Request Broker, or ORB. (That the ORB acronym appears within the CORBA acronym was just too much to be coincidental.) An ORB is a software component whose purpose is to facilitate communication between objects. It does so by providing a number of capabilities, one of which is to locate a remote object, given an object reference. Another service provided by the ORB is the marshaling of parameters and return values to and from remote method invocations. (Don't worry if this explanation doesn't make sense; the ORB is explained in much greater detail on Day 2.) Recall that the Object Management Architecture (OMA) includes a provision for ORB functionality; CORBA is the standard that implements this ORB capability. You will soon see that the use of ORBs provides platform independence to distributed CORBA objects.

Interface Definition Language (IDL)

Another fundamental piece of the CORBA architecture is the use of the Interface Definition Language (IDL). IDL, which specifies interfaces between CORBA objects, is instrumental in ensuring CORBA's language independence. Because interfaces described in IDL can be mapped to any programming language, CORBA applications and components are thus independent of the language(s) used to implement them. In other words, a client written in C++ can communicate with a server written in Java, which in turn can communicate with another server written in COBOL, and so forth.

One important thing to remember about IDL is that it is not an implementation language. That is, you can't write applications in IDL. The sole purpose of IDL is to define interfaces; providing implementations for these interfaces is performed using some other language. When you study IDL more closely on Day 3, you'll learn more about this and other assorted facts about IDL.

The CORBA Communications Model

New Term: CORBA uses the notion of *object references* (which in CORBA/IIOP lingo are referred to as Interoperable Object References, or IORs) to facilitate the communication between objects. When a component of an application wants to access a CORBA object, it first obtains an IOR for that object. Using the IOR, the component (called a *client* of that object) can then invoke methods on the object (called the *server* in this instance).

of the list is unknown in this case, it might be advantageous to use a `sequence`. However, a method cannot return a `sequence` directly; you'll first need to `typedef` a `sequence` of `StockSymbols` to use with this method. For the sake of convenience, add the `typedef` immediately following the `typedef` of the `StockSymbol` type:

```
typedef sequence<StockSymbol> StockSymbolList;
```

You're now ready to add the `getStockSymbols()` method to the `StockServer` interface. This method is described in IDL as follows:

```
StockSymbolList getStockSymbols();
```

That's all the IDL you need for this example. Armed with the `StockMarket.idl` file, you're now ready for the next step: deciding how you'd like to implement these IDL definitions.

Choosing an Implementation Approach

Before actually implementing the server functionality, you'll first need to decide on an implementation approach to use. CORBA supports two mechanisms for implementation of IDL interfaces. Developers familiar with object-oriented concepts might recognize these mechanisms, or at least their names. These include the *inheritance* mechanism, in which a class implements an interface by inheriting from that interface class, and the *delegation* mechanism, in which the methods of the interface class call the methods of the implementing class (delegating to those methods). These concepts are illustrated in Figures 4.1 and 4.2. Figure 4.2 also illustrates that a tie class can inherit from any class—or from no class—in contrast to the inheritance approach, in which the implementation class must inherit from the interface class that it implements.

New Term: Implementation by *inheritance* consists of a base class that defines the interfaces of a particular object and a separate class, inheriting from this base class, which provides the actual implementations of these interfaces.

Implementation by *delegation* consists of a class that defines the interfaces for an object and then delegates their implementations to another class or classes. The primary difference between the inheritance and delegation approaches is that in delegation, the implementation classes need not derive from any class in particular.

A *tie class,* or simply a *tie,* is the class to which implementations are delegated in the delegation approach. Thus, the approach is often referred to as the *tie* mechanism or *tying.*

Most IDL compilers accept command-line arguments to determine which implementation approach to generate code for. Therefore, before you use the IDL compiler to generate code from your IDL definitions, you'll want to determine the approach you want to use. Consult your IDL compiler's documentation to determine which command-line arguments, if any, the IDL compiler expects.

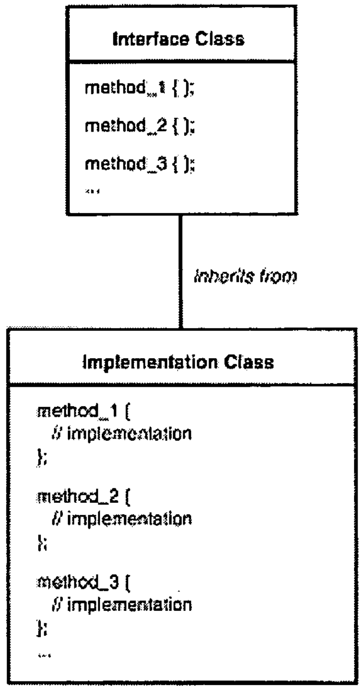

Figure 4.1.
Implementation by inheritance.

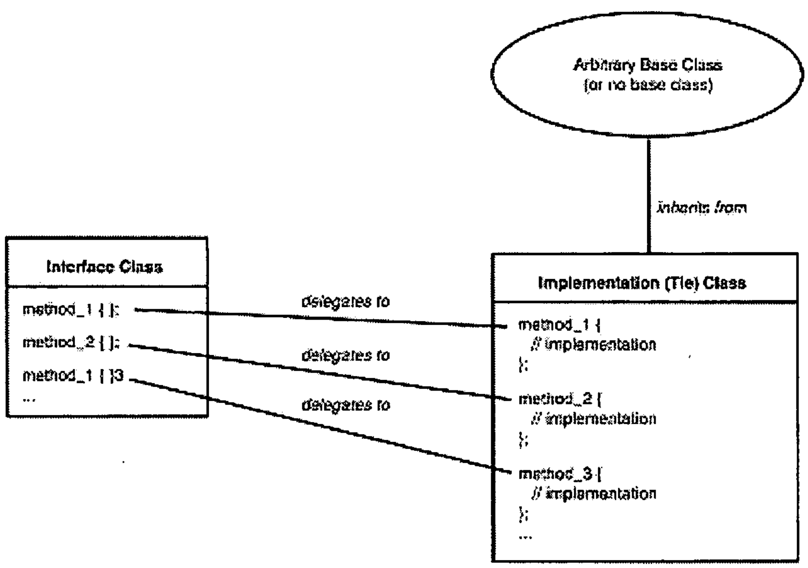

Figure 4.2.
Implementation by delegation.

How to Choose an Implementation Approach

One question you might be asking by now is how to choose an implementation approach. In many cases, this is probably a matter of taste. However, there are certain cases that work well with a particular approach. For example, recall that in the inheritance approach, the implementation class derives from a class provided by the IDL compiler. If an application makes use of legacy code to implement an interface, it might not be practical to change the classes in that legacy code to inherit from a class generated by the IDL compiler. Therefore, for such an application it would make more sense to use the delegation approach; existing classes can readily be transformed into tie classes.

Warning
After you've chosen an implementation approach and have written a great deal of code, be prepared to stick with that approach for that server. Although it's possible to change from one implementation approach to another, this is a very tedious process if a lot of code has already been written. This issue doesn't present itself very often, but you should be aware of it.

For the purposes of this example, either implementation approach will do. The example will use the delegation approach; implementing the server using inheritance will be left as an exercise.

Note that you can usually mix and match the implementation and delegation approaches within a single server application. Although you'll use only one approach per interface, you could choose different approaches for different interfaces in the system. For example, if you had decided that the inheritance approach was the best match for your needs, but you had a few legacy classes that mandated the use of the tie approach, you could use that approach for those classes while using the inheritance approach for the remainder.

Using the IDL Compiler

Now that you have defined your system's object interfaces in IDL and have decided on an implementation approach, you're ready to compile the IDL file (or files, in a more complex system).

Note
The method and command-line arguments for invoking the IDL compiler vary across platforms and products. Consult your product documentation for specific instructions on using your IDL compiler.